;login:

The USENIX Association Newsletter

Volume 12, Number 1 **January/February 1987**

CONTENTS

Call for Papers – Summer 1987 USENIX Conference 3
;login: has New Technical Editor 4
Ten Years Ago in ;login: (a.k.a. UNIX News) 4
How To Write a Setuid Program 5
Matt Bishop
Call for Papers – 4th USENIX Computer Graphics Workshop 12
An Overview of the Sprite Project 13
J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch
Book Review: The C Programmer's Handbook 18
Marc D. Donner
Standards 19
Call for Papers – System Administration Workshop 20
Future Meetings 21
Atlanta Videotapes Available 21
4.3BSD UNIX Manuals 22
4.3BSD Manual Reproduction Authorization and Order Form 23
Proceedings of 3rd Graphics Workshop Available 24
Publications Available 25
Human – Computer Interaction Conference 25
Local User Groups 26

The closing date for submissions for the next issue of *;login:* is February 27, 1987

USENIX **THE PROFESSIONAL AND TECHNICAL UNIX® ASSOCIATION**

NOTICE

;login: is the official newsletter of the USENIX Association, and is sent free of charge to all members of the Association.

The USENIX Association is an organization of AT&T licensees, sub-licensees, and other persons formed for the purpose of exchanging information and ideas about UNIX[†] and similar operating systems and the C programming language. It is a non-profit corporation incorporated under the laws of the State of Delaware. The officers of the Association are:

President	Alan G. Nemeth
Vice-President	Deborah K. Scherrer
Secretary	Waldo M. Wedel
Treasurer	Stephen C. Johnson
Directors	Rob Kolstad Marshall Kirk McKusick John S. Quarterman David A. Yost
Executive Director	Peter H. Salus

The editorial staff of *;login:* is:

Editor and Publisher	Peter H. Salus usenix!peter
Technical Editor	Kevin Baranski-Walker usenix!kevin
Copy Editor	Michelle Dominijanni {masscomp,usenix}!mmp
Production Editor	Tom Strong usenix!strong

Other staff members are:

Betty J. Madden	Office Manager
Emma Reed	Membership Secretary
Jordan Hayes	Technical Consultant

USENIX Association Office
P.O. Box 7
El Cerrito, CA 94530
(415) 528-8649
{ucbvax,decvax}!usenix!office

Judith F. DesHarnais Conference Coordinator

USENIX Conference Office
P.O. Box 385
Sunset Beach, CA 90742
(213) 592-3243 or 592-1381
{ucbvax,decvax}!usenix!judy

John L. Donnelly Exhibit Manager

USENIX Exhibit Office
Oak Bay Building
4750 Table Mesa Drive
Boulder, CO 80303
(303) 499-2600
{ucbvax,decvax}!usenix!johnd

Contributions Solicited

Members of the UNIX community are encouraged to contribute articles to *;login:*. Contributions may be sent to the editors electronically at the addresses above or through the U.S. mail to the Association office. The USENIX Association reserves the right to edit submitted material.

;login: is produced on UNIX systems using *troff* and a variation of the –me macros. We appreciate receiving your contributions in *n/troff* input format, using any macro package. If you contribute hardcopy articles please send **originals** and leave left and right margins of 1″ and a top margin of 1½″ and a bottom margin of 1¼″.

Acknowledgments

The Association uses a VAX[‡] 11/730 donated by the Digital Equipment Corporation for support of office and membership functions, preparation of *;login:*, and other association activities. It runs 4.3BSD, which was contributed, installed, and is maintained by mt Xinu. The VAX uses a sixteen-line VMZ-32 terminal multiplexor donated by Able Computer of Irvine, California.

Connected to the VAX is a QMS Lasergrafix[*] 800 Printer System donated by Quality Micro Systems of Mobile, Alabama. It is used for general printing and draft production of *;login:* with *ditroff* software provided by mt Xinu.

[†]UNIX is a registered trademark of AT&T.

[‡]VAX is a trademark of Digital Equipment Corporation.
[*]Lasergrafix is a trademark of Quality Micro Systems.

Call For Papers
Summer 1987 USENIX Conference
June 8-12, Phoenix, Arizona
Civic Plaza – Convention Center

Abstracts are being accepted from individuals wishing to present papers at the 1987 Summer USENIX Conference. Abstracts should be 250-750 words long, emphasizing what is new and interesting about the work. The final paper should be 8-12 pages when typeset.

Suggested topic areas include, but are not limited to:

- Kernel enhancements, measurements, etc.
- Programming languages and environments.
- UNIX in the office environment.
- Standards and portability.
- *New* mail systems (e.g., X.400-based systems or user interfaces incorporating new interface paradigms).
- Applications, especially unusual ones such as computer aided music, factory automation, etc.
- Security.
- UNIX vs. the naive user.
- Workstations: comparisons, experiences, trends.
- Beyond UNIX – what next?

Vendor presentations should contain substantial technical information and be of interest to the general community.

All abstracts will be due by February 20, 1987. Electronic submissions to *ucbvax!phoenix* or *phoenix@Berkeley.EDU* are preferred. All authors whose abstracts are accepted must submit a paper by the publication deadline or they will forfeit their talk. We currently plan to encourage electronic submissions of final papers in *troff* format using the *-ms* or *-me* macros and the *tbl*, *eqn*, and *pic* preprocessors.

Relevant dates:

20 February	Abstracts due
16 March	Notifications to authors
13 April	Final papers due
10-12 June	Conference program

The program committee consists of:

Eric Allman, Britton Lee (chair)
Greg Chesson, Silicon Graphics
Sam Leffler, Pixar
Jay Lepreau, University of Utah
John Mashey, MIPS
John Quarterman
Dennis Ritchie, AT&T Bell Laboratories
David Taylor, Hewlett-Packard
Chris Torek, University of Maryland

For additional information regarding the program, contact:

Eric Allman
Britton Lee
1919 Addison Suite 105
Berkeley, CA 94704

(415) 548-3211 (work)
ucbvax!eric
(415) 843-9535 (home)
eric@Berkeley.EDU

Please include your network address, if available, with all correspondence. This should be an Arpanet address or a UUCP address relative to a well known host; if in doubt, start from *mcvax*, *ucbvax*, *decvax*, or *seismo*.

;login: has New Technical Editor

After a lengthy search, the USENIX Association is pleased to announce the appointment of Kevin Baranski-Walker as Technical Editor of *;login:*. Kevin will also advise the Executive Director on the establishment of the Association's new quarterly, the first issue of which will probably appear next year.

Kevin has been a systems engineer for the past ten years for a variety of institutions: Science Applications Inc., International Harvester, Electronic Data Systems and Digital Equipment Corp. His claimed expertise has been in real-time and control applications. He adopted UNIX (circa Version 6) as an urgent need to retain some semblance of unity among the half-dozen or so real-time executives being used to moderate the control processor(s). Kevin was involved with ports of Version 7 and System III (massively disfigured) for the Perkin-Elmer 3210 and numerous Intel offerings.

Kevin has worked on applications like airborne side-scanning Infra-Red imaging for location of aquifers, solid state physics approaches to remote data aquisition for geophysical exploration, automatic vehicle transmission control systems, and massively geographically distributed real-time transaction processing.

In addition to the more mundane systems development work, Kevin has kept busy trying his hand at lecturing (at extremely small schools in Chicago) as well as occasionally reviewing for *Communications of the ACM*.

Kevin recently joined the staff of UC Berkeley and feels surprisingly comfortable working "in the public sector." All of us on the Association staff hope that Kevin will feel comfortable working with us.

Kevin can be reached at *usenix!kevin* or *ucbvax!kevin@violet.berkeley.edu*.

– PHS

≡ ≡ ≡

Ten Years Ago in ;login: (a.k.a. UNIX News)

from the February 1977 issue of **UNIX NEWS**

SOFTWARE DISTRIBUTION

Conversations with various people indicate that *stp* runs successfully using the TU16 driver contained in the second software distribution. The bootstrap procedure is to use your current driver and *tp* to get the new TU16 driver and install it in your system before attempting an *stp*.

BUG

The following note was received on the hotline:

We have found a bug in *ttyn*(III). Equality of inodes is determined strictly by inode number. By ignoring the fact that they may be on devices with different major and/or minor dvice numbers. This can cause strange results in some programs, e.g. *goto*(I). Although the fix appears obvious, has anybody already done it? [*Name withheld*]

The fix appeared in **UNIX NEWS** *for May 1977 and will appear in the May 1987 issue of* **;login:**.

– PHS

How To Write a Setuid Program

Matt Bishop
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

UNIX systems allow certain programs to grant privileges to users temporarily; these are called setuid programs. Because they explicitly violate the protection scheme designed into UNIX, they are among the most difficult programs to write. This paper discusses how to write these programs to make using them to compromise a UNIX system as difficult as possible.

Introduction

A typical problem in systems programming is often posed as a problem of keeping records [1]. Suppose someone has written a program and wishes to keep a record of its use. This file, which we shall call the *history file*, must be writable by the program (so it can be kept up to date), but not by anyone else (so that the entries in it are accurate). UNIX solves this problem by providing two sets of identifications for processes. The first set, called the *real* user identification and group identification (or UID and GID, respectively), indicate the real user of the process. The second set, called the *effective* UID and GID, indicate what rights the process has, which may be, and often are, different from the real UID and GID. The protection mask of the file which, when executed, produces the process, contains a bit which is called the *setuid* bit. (There is another such bit for the effective GID.) If that bit is not set, the effective UID of the process will be that of the person executing the file; but if the setuid bit is set (so the program runs in *setuid mode*), the effective UID will be that of the owner of the file, not that of the person executing the file. In either case, the real UID and GID are those of the person executing the file. So if only the owner of the history file (who is the user with the same UID as the file) can write on it, the setuid bit of the file containing the program is turned on, and the UIDs of this file and the history file are the same, then when someone runs the program, that process can write into the history file.

These programs are called *setuid programs*, and exist to allow ordinary users to perform functions which they could not perform otherwise. Without them, many UNIX systems would be quite unusable. An example of a setuid program performing an essential function is a program which lists the active processes on a system with protected memory. Since memory is protected, normally only the privileged user *root* could scan memory to list these processes. However, this would prevent other users from keeping track of their jobs. As with the history file, the solution is to use a setuid program, with *root* privileges, to read memory and list the active processes.

This paper discusses the security problems introduced by setuid programs, and offers suggestions on methods of programming to reduce, or eliminate, these threats. The reader should bear in mind that on some systems, the mere existence of a setuid program introduces security holes; however, it is possible to eliminate the obvious ones.

Attacks

Before we discuss the ways to deal with the security problems, let us look at the two main types of attacks setuid programs can cause. The first involves executing a sequence of commands defined by the attacker (either interactively or via a script), and the second, substituting data of the attacker's choosing for data created by a program.

In the first, an attacker takes advantage of the setuid program's running with special privileges to force it to execute whatever commands he wants. As an example, suppose

an attacker found a copy of the Bourne shell *sh*(1)† that was setuid to *root*. The attacker could then execute the shell, and – since the shell would be interactive – type whatever commands he desired. As the shell is setuid to *root*, these commands would be executed as though *root* had typed them. Thus, the attacker could do anything he wanted, since *root* is the most highly privileged user on the system. Even if the shell were changed to read from a command file (called a *script*) rather than accept commands interactively, the attacker could simply create his own script and run the shell using it. This is an example of something that should be avoided, and sounds like it is easy to avoid – but it occurs surprisingly often.

One way such an attack was performed provides a classic example of why one needs to be careful when designing system programs. A UNIX utility called *at*(1) gives one the capability to have a command file executed at a specified time; the *at* program spools the command file and a daemon executes it at the appropriate time. The daemon determined when to execute the command file by the name under which it was spooled. However, the daemon assumed the owner of the command file was the person who requested that script to be executed; hence, if one could find a world-writable file owned by another in the appropriate directory, one could run many commands with the other's privileges. Cases like this are the reason much of the emphasis on writing good setuid programs involves being very sure those programs do not create world-writable files by accident.

There are other, more subtle, problems with world-writable files. Occasionally programs will use temporary files for various purposes, the function of the program depending on what is in the file. (These programs need not be setuid to anyone.) If the program closes the temporary file at any point and then reopens it later, an attacker can replace the temporary file with a file with other data that will cause the program to act as the attacker desires. If the replacement file has the same owner and group as the temporary file, it can be very difficult for the program to determine if it is being spoofed.

Setuid programs create the conditions under which the tools needed for these two attacks can be made. That does not mean those tools will be made; with attention to detail, programmers and system administrators can prevent an attacker from using setuid programs to compromise the system in these ways. In order to provide some context for discussion, we should look at the ways in which setuid programs interact with their environment.

Threats from the Environment

The term *environment* refers to the milieu in which a process executes. Attributes of the environment relevant to this discussion are the UID and GID of the process, the files that the process opens, and the list of environment variables provided by the command interpreter under which the process executes. When a process creates a subprocess, all these attributes are inherited unless specifically reset. This can lead to problems.

Be as Restrictive as Possible in Choosing the UID and GID

The basic rule of computer security is to minimize damage resulting from a break-in. For this reason, when creating a setuid program, it should be given the least dangerous UID and GID possible. If, for example, game programs were setuid to *root*, and there were a way to get a shell with *root* privileges from within a game, the game player could compromise the entire computer system. It would be far safer to have a user called *games* and run the game programs setuid to that user. Then, if there were a way to get a shell from within a game, at worst it would be setuid to *games* and only game programs could be compromised.

Related to this is the next rule.

Reset Effective UIDs Before Calling exec‡

† The usual notation for referencing UNIX commands is to put the name of the command in italics, and the first time the name appears in a document, to follow it by the section number of the *UNIX Programmers' Manual* in which it appears; this number is enclosed in parentheses. There are two versions of the manual referred to in this paper, one for 4.2BSD UNIX [2], and one for System V UNIX [3]. Most commands are in the same section in both manuals; when this is not true, the section for each manual will be given.

‡ *Exec* is a generic term for a number of system and library calls; these are described by the manual pages *exec*(2) in the System V manual and *execve*(2) and *execl*(3) in the 4.2BSD manual.

Resetting the effective UID and GID before calling *exec* seems obvious, but it is often overlooked. When it is, the user may find himself running a program with unexpected privileges. This happened once at a site which had its game programs setuid to *root*; unfortunately, some of the games allowed the user to run subshells from within the games. Needless to say, this problem was fixed the day it was discovered!

One difficulty for many programmers is that *exec* is often called within a library subroutine such as *popen*(3) or *system*(3) and that the programmer is either not aware of this, or forgets that these functions do not reset the effective UIDs and GIDs before calling *exec*. Whenever calling a routine that is designed to execute a command as though that command were typed at the keyboard, the effective UID and GID should be reset unless there is a specific reason not to.

Close All But Necessary File Descriptors Before Calling exec

This is another requirement that most setuid programs overlook. The problem of failing to do this becomes especially acute when the program being *exec*'ed may be a user program rather than a system one. If, for example, the setuid program were reading a sensitive file, and that file had descriptor number 9, then any *exec*'ed program could also read the sensitive file (because, as the manual page warns, "[d]escriptors open in the calling process remain open in the new process ...").

The easiest way to prevent this is to set a flag indicating that a sensitive file is to be closed whenever an *exec* occurs. The flag should be set immediately after opening the file. Let the sensitive file's descriptor be *sfd*. In both System V and 4.2BSD, the system call

fcntl(*sfd*, F_SETFD, 1)

will cause the file to close across *exec*'s; in both Version 7 and 4.2BSD, the call

ioctl(*sfd*, FIOCLEX, NULL)

will have the same effect. (See *fcntl*(2) and *ioctl*(2) for more information.)

Be Sure a Restricted Root Really Restricts

The *chroot*(2) system call, which may be used only by *root*, will force the process to treat the argument directory as the root of the file system. For example, the call

chroot("/usr/riacs")

makes the root directory */usr/riacs* so far as the process which executed the system call is concerned. Further, the entry '..' in the new root directory is interpreted as naming the root directory. Where symbolic links are available, they too are handled correctly.

However, it is possible for *root* to link directories just as an ordinary user links files. This is not done often, because it creates loops in the UNIX file system (and that creates problems for many programs), but it does occasionally occur. These directory links can be followed regardless of whether they remain in the subtree with the restricted root. To continue the example above, if */usr/demo* were linked to */usr/riacs/demo*, the sequence of commands

cd /demo
cd ..

would make the current working directory be */usr*. Using relative path names at this point (since an initial '/' is interpreted as */usr/riacs*), the user could access any file on the system. Therefore, when using this call, one must be certain that no directories are linked to any of the descendants of the new root.

Check the Environment In Which the Process Will Run

The environment to a large degree depends upon certain variables which are inherited from the parent process. Among these are the variables **PATH** (which controls the order and names of directories searched by the shell for programs to be executed), **IFS** (a list of characters which are treated as word separators), and the parent's *umask*, which controls the protection mode of files that the subprocess creates.

One of the more insidious threats comes from routines which rely on the shell to execute a program. (The routines to be wary of here are *popen*, *system*, *execlp*(3), and *execvp*(3).†) The danger is that the shell will not execute the program intended. As an example, suppose a program that is setuid to *root* uses *popen* to execute the program

† *execlp* and *execvp* are in section 2 of the System V manual.

printfile. As *popen* uses the shell to execute the command, all a user needs to do is to alter his **PATH** environment variable so that a private directory is checked before the system directories. Then, he writes his own program called *printfile* and puts it in that private directory. This private copy can do anything he likes. When the *popen* routine is executed, his private copy of *printfile* will be run, with *root* privileges.

On first blush, limiting the path to a known, safe path would seem to fix the problem. Alas, it does not. When the Bourne shell *sh* is used, there is an environment variable **IFS** which contains a list of characters that are to be treated as word separators. For example, if **IFS** is set to 'o', then the shell command *show* (which prints mail messages on the screen) will be treated as the command *sh* with one argument *w* (since the 'o' is treated as a blank). Hence, one could force the setuid process to execute a program other than the one intended.

Within a setuid program, all subprograms should be invoked by their full path name, or some path known to be safe should be prefixed to the command; and the **IFS** variable should be explicitly set to the empty string (which makes white space the only command separators).

The danger from a badly-set *umask* is that a world-writable file owned by the effective UID of a setuid process may be produced. When a setuid process must write to a file owned by the person who is running the setuid program, and that file must not be writable by anyone else, a subtle but nonetheless dangerous situation arises. The usual implementation is for the process to create the file, *chown*(2) it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chown*'ed the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

There are two ways to prevent this; the first is fairly simple, but requires the effective UID to be that of *root*. (The other method does not suffer from this restriction; it is described in a later section.) The *umask*(2) system call can be used to reset the *umask* within the setuid process so that the file is at no time world-writable; this setting overrides any other, previous settings. Hence, simply reset *umask* to the desired value (such as 022, which prevents the file from being opened for writing by anyone other than the owner) and then open the file. (The *umask* can be reset afterwards without affecting the mode of the opened file.) Upon return, the process can safely *chown* the file to the real UID and GID of the process. (Incidentally, only *root* can *chown* a file, which is why this method will not work for programs the effective UID of which is not *root*.) Note that if the process is interrupted between the *open*(2) and the *chown* the resulting file will have the same UID and GID as the process' effective UID and GID, but the person who ran the process will not be able to write to that file (unless, of course, his UID and GID are the same as the process' effective UID and GID).

As a related problem, *umask* is often set to a dangerous value by the parent process; for example, if a daemon is started at boot time (from the file */etc/rc* or */etc/rc.local*), its default *umask* will be 0. Hence, any files it creates will be created world-writable unless the protection mask used in the system call creating the file is set otherwise.

Programming Style

Although threats from the environment create a number of security holes, inappropriate programming style creates many more. While many of the problems in programming style are fairly typical (see [4] for a discussion of programming style in general), some are unique to UNIX and some to setuid programs.

Do Not Write Interpreted Scripts That Are Setuid

Some versions of UNIX allow command scripts, such as shell scripts, to be made setuid. This is done by applying the setuid bit to the command interpreter used, and then interpreting the commands in the script. Unfortunately, given the power and complexity of many command interpreters, it is often possible to force them to perform actions which were not intended, and which allow the user to violate system security. This leaves the owner of the setuid script open to a devastating attack. In general, such scripts should be avoided.

As an example, suppose a site has a setuid script of *sh* commands. An attacker simply executes the script in such a way that the shell which interprets the commands appears to have been invoked by a person logging in. UNIX applies the setuid bit on the script to the shell, and since it appears to be a login shell, it becomes interactive. At that point, the attacker can type his own commands, regardless of what is in the script.

One way to avoid having a setuid script is to turn off the setuid bit on the script, and rather than calling the script directly, use a setuid program to invoke it. This program should take care to call the command interpreter by its full path name, and reset environment information such as file descriptors and environment variables to a known state. However, this method should be used only as a last resort and as a temporary measure, since with many command interpreters it is possible even under these conditions to force them to take some undesirable action.

Do Not Use creat *for Locking*

According to its manual page, "The mode given [*creat*(2)] is arbitrary; it need not allow writing. This feature has been used ... by programs to construct a simple exclusive locking mechanism." In other words, one way to make a lock file is to *creat* a file with an unwritable mode (mode 000 is the most popular for this). Then, if another user tried to *creat* the same file, *creat* would fail, returning −1.

The only problem is that such a scheme does not work when at least one of the processes has *root*'s UID, because protection modes are ignored when the effective UID is that of *root*. Hence, *root* can overwrite the existing file regardless of its protection mode. To do locking in a setuid program, it is best to use *link*(2). If a link to an already-existing file is attempted, *link* fails, even if the process doing the linking is a *root* process and the file is not owned by *root*.

With 4.2 Berkeley UNIX, an alternative is to use the *flock*(3) system call, but this has disadvantages (specifically, it creates advisory locks only, and it is not portable to other versions of UNIX).

The issue of covert channels [5] also arises here; that is, information can be sent illicitly by controlling resources. However, this problem is much broader than the scope of this paper, so we shall pass over it.

Catch All Signals

When a process created by running a setuid file dumps core, the core file has the same UID as the real UID of the process.† By setting *umask*'s properly, it is possible to obtain a world-writable file owned by someone else. To prevent this, setuid programs should catch all signals possible.

Some signals, such as **SIGKILL** (in System V and 4.2BSD) and **SIGSTOP** (in 4.2BSD), cannot be caught. Moreover, on some versions of UNIX, such as Version 7, there is an inherent race condition in signal handlers, When a signal is caught, the signal trap is reset to its default value and *then* the handler is called. As a result, receiving the same signal immediately after a previous one will cause the signal to take effect regardless of whether it is being trapped. On such a version of UNIX, signals cannot be safely caught. However, if a signal is being *ignored*, sending the process a signal will *not* cause the default action to be reinstated; so, signals can be safely ignored.

The signals **SIGCHLD**, **SIGCONT**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**‡ (all of which relate to the stopping and starting of jobs and the termination of child processes) should be caught unless there is a specific reason not to do this, because if data is kept in a world-writable file, or data or lock files in a world-writable directory such as */tmp*, one can easily change information the process (presumably) relies upon. Note, however, that if a system call which creates a child (such as *system*, *popen*, or *fork*(2)) is used, the **SIGCHLD** signal will be sent to the process when the command given *system* is finished; in this case, it would be wise to ignore **SIGCHLD**.

† On some versions of UNIX, such as 4.2BSD, no core file is produced if the real and effective UIDs of the process differ.

‡ The latter four are used by various versions of Berkeley UNIX and their derivatives to suspend and continue jobs. They do not exist on many UNIXes, including System V.

This brings us to our next point.

Be Sure Verification Really Verifies

When writing a setuid program, it is often tempting to assume data upon which decisions are based is reliable. For example, consider a spooler. One setuid process spools jobs, and another (called the *daemon*) runs them. Assuming that the spooled files were placed there by the spooler, and hence are "safe," is again a recipe for disaster; the *at* spooler discussed earlier is an example of this. Rather, the daemon should attempt to verify that the spooler placed the file there; for example, the spooler should log that a file was spooled, who spooled it, when it was spooled, and any other useful information, in a protected file, and the daemon should check the information in the log against the spooled file's attributes. With the problem involving *at*, since the log file is protected, the daemon would never execute a file not placed in the spool area by the spooler.

Make Only Safe Assumptions About Recovery Of Errors

If the setuid program encounters an unexpected situation that the program cannot handle (such as running out of file descriptors), the program should not attempt to correct for the situation. It should stop. This is the opposite of the standard programming maxim about robustness of programs, but there is a very good reason for this rule. When a program tries to handle an unknown or unexpected situation, very often the programmer has made certain assumptions which do not hold up; for example, early versions of the command *su*(1) made the assumption that if the password file could not be opened, something was disastrously wrong with the system and the person should be given *root* privileges so that he could fix the problem. Such assumptions can pose extreme danger to the system and its users.

When writing a setuid program, keep track of things that can go wrong – a command too long, an input line too long, data in the wrong format, a failed system call, and so forth – and at each step ask, "if this occurred, what should be done?" If none of the assumptions can be verified, or the assumptions do not cover all cases, at that point the setuid program should *stop*. Do not attempt to recover unless recovery is guaranteed; it is too easy to produce undesirable side-effects in the process.

Once again, when writing a setuid program, if you are not sure how to handle a condition, exit. That way, the user cannot do any damage as a result of encountering (or creating) the condition.

For an excellent discussion of error detection and recovery under UNIX, see [6].

Be Careful With I/O Operations

When a setuid process must create and write to a file owned by the person who is running the setuid program, either of two problems may arise. If the setuid process does not have permission to create a file in the current working directory, the file cannot be created. Worse, it is possible that the file may be created and left writable by anyone. The usual implementation is for the process to create the file, *chown* it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chown*'ed, the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

The section on checking the environment described a method of dealing with this situation when the program is setuid to *root*. That method does not work when the program is setuid to some other user. In that case, the way to prevent a setuid program from creating a world-writable file owned by the effective UID of the process is far more complex, but eliminates the need for any *chown* system calls. In this method, the process *fork*(2)'s, and the child resets its effective UID and GID to the real UID and GID. The parent then writes the data to the child via *pipe*(2) rather than to the file; meanwhile, the child creates the file and copies the data from the pipe to the file. That way, the file is never owned by the user whose UID is the effective UID of the setuid process.

Some UNIX systems, notably 4.2BSD, allow a third method. The basic problem here is that the system call *setuid*(3)† can only set the effective UID to the real UID (unless the process runs with *root* privileges, in which

† This system call is in section 2 of the System V manual.

case both the effective and real UIDs are reset to the argument). Once the effective UID is reset with this call, the old effective UID can never be restored (again, unless the process runs with *root* privileges). So it is necessary to avoid resetting any UIDs when creating the file; this leads to the creation of another process or the use of *chown*. However, 4.2BSD provides the capability to reset the effective UID independently of the real UID using the system call *setreuid*(2). A similar call, *setregid*(2), exists for the real and effective GIDs. So, all the program need do is use these calls to exchange the effective and real UIDs, and the effective and real GIDs. That way, the old effective UID can be easily restored, and there will not be a problem creating a file owned by the person executing the setuid program.

Conclusion

To summarize, the rules to remember when writing a setuid program are:

- Be as restrictive as possible in choosing the UID and GID.
- Reset effective UIDs and GIDs before calling *exec*.
- Close all but necessary file descriptors before calling *exec*.
- Be sure a restricted root really restricts.
- Check the environment in which the process will run.
- Do not write interpreted scripts that are setuid.
- Do not use *creat* for locking.
- Catch all signals.
- Be sure verification really verifies.
- Make only safe assumptions about recovery of errors.
- Be careful with I/O operations.

Setuid programs are a device to allow users to acquire new privileges for a limited amount of time. As such, they provide a means for overriding the protection scheme designed into UNIX. Unfortunately, given the way protection is handled in UNIX, it is the best solution possible; anything else would require users to share passwords widely, or the UNIX kernel to be rewritten to allow access lists for files and processes. For these reasons, setuid programs need to be written to keep the protection system as potent as possible even when they evade certain aspects of it. Thus, the designers and implementors of setuid programs should take great care when writing them.

Acknowledgements

Thanks to Bob Brown, Peter Denning, George Gobel, Chris Kent, Rich Kulawiec, Dawn Maneval, and Kirk Smith, who reviewed an earlier draft of this paper, and made many constructive suggestions.

References

[1] Aleph-Null, "Computer Recreations," *Software – Practise and Experience* **1**(2) pp. 201-204 (April–June 1971).

[2] *UNIX System V Release 2.0 Programmer Reference Manual*, DEC Processor Version, AT&T Technologies (April 1984).

[3] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (August 1983).

[4] Kernighan, Brian and Plauger, P., *The Elements of Programming Style, Second Edition*, McGraw-Hill Book Company, New York, NY (1978).

[5] Lampson, Butler, "A Note on the Confinement Problem," *CACM* **16**(10) pp. 613-615 (October 1973).

[6] Darwin, Ian and Collyer, Geoff, "Can't Happen or /* NOTREACHED */ or Real Programs Dump Core," 1985 Winter USENIX Proceedings (January 1985).

Call for Papers
4th USENIX Computer Graphics Workshop

The 4th USENIX Workshop on Computer Graphics and Similar Activities will be held on October 8-9, 1987 in Cambridge, Mass. Participation and papers are solicited in all areas of computer graphics, computer-aided geometric design, and interactive techniques including:

Animation and motion control
Document description languages
Font design and display
Page description languages
Geometric Modelling
Image synthesis
Interactive display or sound systems
Paint systems
Picture storage, compression and transmission
Real-time audio and/or video processing or control
Windowing systems

Attendance at the workshop will be limited. Presentations will be scheduled to allow significant time for discussion.

Deadlines:

Abstracts and position papers must be received by May 1, 1987.

Participants will be notified of acceptance by June 1, 1987.

Completed papers, slides, videotapes for inclusion in the proceedings must be received by August 14, 1987.

Abstracts and enquiries regarding the technical program should be sent to:

Tom Duff, Program Chairman
Bell Laboratories
Room 2C-425
200 Mountain Avenue
Murray Hill, NJ 07974
(201) 582-6485
{usenix|ucbvax|decvax|attmail}!research!td

An Overview of the Sprite Project

John Ousterhout
Andrew Cherenson
Fred Douglis
Michael Nelson
Brent Welch

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

spriters@arpa.berkeley.edu

Introduction

Sprite is a new operating system that we have been designing and implementing at U.C. Berkeley over the last two years. The design was strongly influenced by three pieces of technology: local-area networks, large physical memories, and multiprocessor workstations. This article is a brief summary of how the Sprite design reflects those technologies.

The single most important issue for us is the network. We hope to provide simple and efficient mechanisms for a collection of workstations (nodes) to work together over a local-area network or small internet. Ideally, all of the resources of the collection of nodes (files, devices, CPU cycles, etc.) should be transparently and efficiently accessible by each of the nodes. We would like to achieve an effect something like timesharing in its ease of sharing, but with the high performance and guaranteed interactive response that personal workstations can provide. We'd like Sprite to support communities of a few dozen to a few hundred people; growth beyond this size is less important to us than the "quality of life" we provide for smaller communities. Sprite provides two network-oriented features: a network file system and a process migration facility.

The second piece of technology that influenced Sprite's design is the ever-increasing amout of physical memory available on workstations. Sprite takes advantage of large memories with large file caches, both on clients and servers. The nodes use a simple consistency protocol to coordinate shared access to files. The file caches adjust their sizes dynamically, trading memory back and forth with the virtual memory system.

The third technological influence on Sprite is multiprocessors, which appear likely to become available in workstation configurations within a few years. To support multiprocessors, Sprite implements shared address spaces and has a multi-threaded kernel.

Basics

There are several other experimental or commercial operating systems that share some of Sprite's goals, particularly those related to networks. These include LOCUS [6], V [2], Accent [3], and systems marketed by Sun and Apollo. Of these systems, Sprite is most like LOCUS and Sun's version of UNIX, in that the kernel-call interface provided to user programs is almost identical to that of 4.3 BSD UNIX. However, while the kernels used by LOCUS and Sun are based on early versions of the BSD kernel, Sprite's kernel has been rebuilt from scratch. The Sprite kernel differs from the BSD-derived kernels in two major ways: it is based on a general-purpose kernel-to-kernel remote procedure call package; and it is multi-threaded.

The most important feature of the Sprite kernel implementation is its remote-procedure-call (RPC) facility, which the kernels of different nodes use to request services of each other [7]. The lightweight connection/acknowledgement protocol in Sprite's RPC mechanism is similar to the scheme of Birrell and Nelson [1]. Each kernel contains a small set of server processes that respond to incoming requests (the exact number of processes varies dynamically in response to the machine's RPC load). We have not yet implemented any particular security mechanisms: each of the kernels trusts each of the other kernels. Since we did not have

access to a stub generator, the server and client stubs for each call were written by hand (there are a few dozen of these).

The basic round-trip time for a simple RPC with no parameters is about 5 ms on Sun-2 workstations. Unlike Birrell and Nelson's scheme, Sprite's RPC provides for fragmentation of large packets. This allows large blocks of data to be transferred between kernels at more than 300 kbytes/sec. on Sun-2 workstations, which is 2 to 3 times faster than we initially achieved without fragmentation. See [7] for a detailed description and performance analysis of Sprite's RPC mechanism.

Our second major kernel change was to support multiprocessors. The BSD kernels are single-threaded: a process executing in the kernel cannot be preempted and the system code depends on the fact that a process executing in the kernel has exclusive access to all of the kernel data structures. In a multiprocessor configuration, it is not safe for more than one process to be executing kernel code at a time. In Sprite, we used a monitor-like approach, with explicit locks on modules and data structures. This allows more than one process to execute kernel code at the same time, which will be important when multiprocessor workstations become available.

The Network File System

The network file system is the heart of Sprite. As in most timeshared UNIX systems, the Sprite file system provides a shared mechanism for naming, for long-term storage, and for interprocess communication. To a network of Sprite workstations, there appears to be a single hierarchical UNIX-like file system with a single shared root. The same name yields the same file on every node of the system. The basic operations on Sprite files (open, close, read, write, seek, ioctl, lock, unlock, etc.) behave the same as if they were executed on a single BSD timesharing system, instead of a network of workstations. This differs from Sun's NFS, which does not support operations requiring state to be maintained on the server (e.g. lock and unlock).

The three most unusual aspects of the Sprite file system are its use of *prefix tables* for server location, its implementation of client-level caching, and its named pipes.

In any network file system with multiple servers, there must be a mechanism to determine which server is responsible for which files. Typically, each server handles one or more subtrees of the file hierarchy. In the LOCUS and NFS file systems, each client maintains a static *mount table* that associates servers with subtrees. There are no automatic mechanisms to update mount tables when server configurations change; as a consequence, large network systems are difficult to administer. In contrast, Sprite uses a more dynamic form of table called a *prefix table*. Each client maintains a small table that associates servers with subtrees using prefixes. When opening a file, the first few characters of the file's name are matched against each of the prefixes in the table; the open request is then sent to the server associated with the longest matching prefix. The prefix tables are created and modified dynamically using a simple broadcast protocol. There are no static global tables identifying the system configuration; all that is needed is for each server to know which prefixes it serves and to respond to broadcasts for information about those prefixes. Prefix tables are also flexible enough to handle private local subtrees and even a simple form of replication; see [8] for details.

In Sprite, both clients and servers keep main-memory caches of recently-used disk blocks. As memories get larger and larger, we expect these caches to get larger and larger too. Within a few years, cache hit ratios of nearly 100% should be common (see [5] for trace-driven predictions of cache effectiveness). File-system block caches have two advantages in a network operating system. First, they reduce disk traffic; our preliminary measurements suggest that this alone can result in factors of 2 to 5 in file-system throughput. Second, they eliminate network traffic between clients and servers, which provides an additional performance improvement.

The main problem with client file caches is cache consistency. If multiple clients are reading and writing a file at the same time, each read request must return the most recent information written to the file, regardless of which nodes are making read and write requests. Sprite guarantees cache consistency by disabling client caching for any file that is open for writing on one node while also open for reading or writing on some other node.

Consistency is implemented by the servers, with consistency actions being taken when files are opened or closed. Each server keeps track of which of its files are open on which nodes; when a server receives an open request that will result in conflicting access to a file, client caching (for that one file) is disabled on all nodes and the file's dirty blocks (if any) are flushed back from client caches to the server. This scheme permits caching under multiple-reader access, and also allows unshared files with short lifetimes to be created, used, and deleted entirely within a client cache, without ever being written to the server.

Although Sprite's approach suffers a performance degradation when consistency conflicts cause client caching to be disabled for a file, our measurements of current UNIX systems indicate that files tend to be open only for very short intervals (reducing the likelihood of a conflict), and that write-sharing of files is extremely infrequent [5]. An alternate approach is that of LOCUS, which is based on token-passing. One node (the current owner of the file's token) is permitted to cache blocks of a file, even when the file is open by multiple readers and writers. The other nodes must wait until they receive the token before accessing the file. In comparison to Sprite's mechanism, the token-passing approach requires extra implementation complexity and seems unlikely to reduce server traffic, since client caches must be flushed whenever the token changes hands.

In addition to the standard UNIX file facilities, Sprite provides *named pipes*, which combine the features of traditional UNIX files and pipes. We expect named pipes to be used instead of TCP/IP for inter-process communication within a Sprite system. A named pipe has a name in the file system, occupies space on disk, and persists even when not open by any processes. However, it has the operations of a normal pipe. Reads return the oldest data in the pipe and remove that data from the pipe atomically. Writes atomically append data to the end of the pipe. Processes attempting to read from an empty named pipe are suspended until data is written to the pipe. Sprite named pipes are similar to LOCUS named pipes, except that a Sprite named pipe retains its information even when not open (in LOCUS, un-opened named pipes are always empty). Client and server caches are used to avoid writing pipe information to disk unless it is long-lived; we expect this to result in performance at least as good as TCP/IP.

Virtual Memory

In comparison to the BSD implementation of virtual memory, there are two major changes in Sprite: shared address spaces and backing files.

In anticipation of multiprocessor workstations, and to support closely-cooperating teams of processes, Sprite provides a simple form of address-space sharing. As in traditional UNIX, each process's address space consists of three segments: read-only code, writable heap, and writable stack. In UNIX, however, only the code segment may be shared between processes. In Sprite, both code and heap may be shared. Stacks are still private. An argument to the *fork* system call determines whether or not the new process will share the heap of the old process. This mechanism provides all-or-nothing sharing. It is suitable for a collection of processes all working on the same application, but does not allow a process to share one area of memory with one set of processes and a different area with a different set of processes. The Sprite scheme is simpler than the region-based memory mechanisms being discussed for 4.4 BSD, but less powerful.

UNIX has traditionally used a special area of disk for paging storage, with special-purpose algorithms for managing the paging storage. In contrast, Sprite uses ordinary files for paging storage. Each of a process's three segments has a corresponding backing file. For the code, it is the binary file from which the program was loaded (unless the code is to be writable); for the stack and heap, temporary files are allocated in a special directory of the file system. Ordinary file operations are used to read and write the backing files; storage for the backing files is not allocated until pages are actually written out [4].

Sprite's use of backing files has several advantages over the traditional UNIX approach. First, it is simpler, since no special algorithms need to be implemented for managing backing store. Second, by using the network file system for backing files, the paging information can easily be accessed by all the nodes on the network; this is important for process migration (see below). Third, it allows

the server's disk cache to serve as an extended memory for the clients it serves. Fourth, it saves disk space since backing storage is only allocated when needed. In contrast, the BSD-based implementations of virtual memory allocate statically a separate area of disk for each workstation to use for backing storage. The result is that backing storage occupies a huge amount of disk space on the file servers, with very little of it in actual use at any given time. With the large block sizes and clever disk allocation mechanisms used by modern file systems, we don't see any performance advantage to using a special-purpose paging store.

Virtual Memory vs. File-System Cache

Both the virtual-memory system and the file cache require physical memory, and each would like to use as much memory as possible. In traditional UNIX, the file cache is fixed in size at system boot time, and the virtual-memory system uses what's left. For Sprite, we decided to permit the boundary between virtual memory and file system to change dynamically. When not much memory is needed for virtual memory, the file cache can grow to fill the unused memory; when more memory is needed for VM, the file cache shrinks. This allows applications with small virtual memory needs (such as the compiler and editor) to use almost all of memory as a large file cache, while permitting larger applications to use all of memory for the pages of their address spaces.

To implement the flexible boundary between the file cache and VM, each component manages its pages using an LRU-like algorithm (perfect LRU for the file cache, clock for VM). When VM needs a new page, it compares the age of its oldest page with the age of the file cache's oldest page. If the file cache's page is older, then the file cache frees up a page and gives it to VM. A similar approach is used by the file cache with it needs a new page.

Process Migration

Sprite allows processes to be migrated between nodes with compatible instruction sets. Process migration is simplified by the availability of a shared network file system (which allows the process to continue to access the same files from its new node), and the use of files for backing store. To migrate a process from node A to node B, node A writes out all of the process's dirty pages to the file server and transfers the execution state to node B. Node B then reloads the process's pages from the file server on a demand basis as the process executes. The file-system cache-consistency protocols guarantee that any file modifications made by the process on node A will also be visible to the process on node B.

The cost of process migration is determined primarily by the number of dirty pages in its address space. On Sun-2 workstations, an empty process can be migrated in about 0.5 second, with dirty pages flushed at about 100 kbytes/sec.

Although many system calls (e.g. those related to the file system) are already location-independent, there are others that may have different effects if executed on different machines (e.g. "get time of day"). To handle the non-transparent system calls, Sprite assigns to each process a *home node*; location-sensitive kernel calls are sent to the home node using the kernel-to-kernel RPC mechanism, and are processed on the home node. This guarantees that a process produces exactly the same results whether or not it has been migrated. Although sending system calls home is expensive, our preliminary measurements indicate that this happens infrequently, and that migrated processes suffer no more than a 5-10% performance degradation.

Although the basic process migration mechanism is now running, we haven't yet implemented the policy mechanisms to accompany it. We plan to provide facilities akin to the '&', 'bg', and 'fg' facilities of *csh*, which will offload a process to an idle workstation instead of putting it in background. Sprite will keep track of which workstations are idle (e.g. those whose users have not been active for at least 10 minutes) and choose one of them for the offloading. If a user returns to a workstation on which processes have been offloaded, the foreign processes will be ejected, either back to their home node or to another idle node. We are also implementing a new version of the *make* utility called *pmake*, which uses the process migration facilities to rebuild large systems in parallel.

Project Status

The Sprite team was formed in the fall of 1984, and we began coding in the spring of 1985. As of today, all of the major pieces of the system are operational except for the site-selection mechanisms of process migration and some recovery code. We are currently in the phase of the project where bugs are appearing at least as fast as we can fix them, but we hope that the system will stabilize enough for us to start using it for everyday work sometime in the next several months. We expect to have detailed performance measurements available within a few months. At this point all we can say is that our initial measurements on the untuned system are very encouraging (aren't they always?). It's still too early to make any conclusions about the success of our design decisions.

Other Operating-System Work at Berkeley

Sprite is not the only operating system project underway at Berkeley. Two others that you may also be interested in are the DASH project and the 4.n BSD work. DASH is a project being led by Profs. David Anderson and Domenico Ferrari. The letters of the name stand for "Distributed," "Autonomous," "Secure," and "Heterogenous"; the DASH project is exploring these issues in the context of very large internetworks (whereas Sprite is concerned primarily with tightly-cooperating, mutually-trusting, homogeneous groups of machines that are not widely-distributed). The 4.n BSD work, in which Mike Karels and Kirk McKusick are the principals, should need no further introduction to this community; it is continuing the evolution of Berkeley UNIX with new features such as region-based memory management and a more flexible file system structure to permit multiple network file systems to co-exist.

References

[1] Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.

[2] Cheriton, D. R. "The V Kernel: A Software Base for Distributed Systems." *IEEE Software*, Vol. 1, No. 2, April 1984, pp. 19-42.

[3] Fitzgerald, R. and Rashid, R. F. "The Integration of Virtual Memory Management and Interprocess Communication in Accent." *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986, pp. 147-177.

[4] Nelson, M. *Virtual Memory for the Sprite Operating System*, Technical Report UCB/CSD 86/301, Computer Science Division, University of California, Berkeley, June 1986.

[5] Ousterhout, J. K. et al. "A Trace-Driven Analysis of the 4.2 BSD UNIX File System." Proceedings of the 10th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 19, No. 5, December 1985, pp. 15-24.

[6] Walker, B. et al. "The LOCUS Distributed Operating System." Proceedings of the 9th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 17, No. 5, November 1983, pp. 49-70.

[7] Welch, B. *The Sprite Remote Procedure Call System*, Technical Report UCB/CSD 86/302, Computer Science Division, University of California, Berkeley, June 1986.

[8] Welch, B. and Ousterhout, J. "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System." *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 184-189.

Book Review

The C Programmer's Handbook by M. I. Bolsky (AT&T Systems Center)

(New York: Prentice-Hall, Inc., 1985) $16.95

Reviewed by Marc D. Donner

IBM Thomas J. Watson Research Center
ucbvax!ibm.com!donner

Quick! What is the order of arguments of *ungetc*? Is it *ungetc(char, stream)* or is it *ungetc(stream, char)*? Well, you grab your copy of K&R, go to the index. Hmm, let's see, that's pretty far down. Ah, here it is in the index on page 228. Now you thumb through the book looking for page 156 ... and here it is near the bottom ... *ungetc(c, fp)*.†

On the other hand, I grab my copy of *The C Programmer's Handbook* and turn it over and look at the index to library functions printed on the back cover. It tells me that *ungetc* is documented on page 49. I flip the book open to page 49, quite easy because the numbers are printed in a large font at the outside upper corner of each page, the pages are thick, and there aren't too many in the handbook. And there it is, at the bottom, *ungetc(c, stream)*. And it tells me that *c* is an `int` and that *stream* is a `FILE*`!

There are many unusual things about this handbook, and all of them contribute to making it one of the most usable books I possess. On the front cover is printed the table of contents. The back cover has the index to library functions plus a list of all of the operators in precedence order, with notations about whether they associate right-to-left or left-to-right. The book is spiral bound so that it lies flat, a big plus when you want to refer to something while hacking. How much would you pay for a spiral bound copy of K&R? The pages are uncrowded, making the information more easily locatable. The format is simple and consistent throughout, again contributing to ease of use. The paper is unusual ... it is fairly thick and of high quality, making the resulting pages easy to handle. Normally, thick paper is low quality, but this book seems to be an exception.

Each entry in the book is introduced by a red headline. Structurally each entry is an unordered list, generally containing some syntax, some explanation, an example when appropriate, and a reference to the section in the on-line manual pages where more detail resides. The designers weren't intimidated by white space, and they use it to considerable advantage, making the entries easy to read and find.

One minor drawback is the choice of sanserif fonts for the printing. It is well known that serifed fonts are more readable, though sanserif fonts are widely believed to be more "modern." Another, more serious problem, is the fact that this handbook is specific to System V. I have been using it while programming my 4.2 system for several months now and the only mismatch that has really bothered me is that the 4.2 *index* function is called *strchr* in the System V library. I find the name *index* to be more informative, though I can certainly understand the choice of *strchr*. In an ideal world, the book would be available in a 4.2 version, also.

I suppose that reviewing a book like this is like reviewing the telephone book ... a bit thin on plot, but interesting typesetting. There is almost no writing in this book, it is the closest thing to a real handbook that I have ever seen going under that name. This book is deceptively simple, but a great deal of thought must have gone into each decision. Even such choices as the paper it is printed on seem to have received careful attention.

Please don't be overwhelmed by my enthusiasm, this book is not the be-all and end-all, but it is very nice to see something simple like this done well once in a while. Something that doesn't pretend to be much, but is a most excellent example of what it is.

† By the way, I asked this question of a prominent UNIX guru of my acquaintance while writing this review and he proposed two answers, both wrong; so this is not a trivial question.

Standards

Ima Tired

You know what I think? I think that people are not taking this standards thing very seriously.

Standards are important things, you know. Ignoring Gregorian chants, Ben Franklin was among the first to use standards. He recommended the use of interchangeable parts in rifles. This reduced their downtime, increased their performance, and considerably increased their repairability.

Imagine living in the olden times and breaking a part on your rifle:

He: Hey Rachel, the flintlock's broken.

She: Bad news, Harry. Our gunsmith, Withers, passed away last year. You're going to have to get a whole new gun.

He: You mean no one has a new flintlock?

She: Nope, they're one-of-a-kind. Only Withers knew how ours worked.

Ben had many good ideas. Henry Ford carried them to perfection.

So where are we now? There are standards everywhere, you know. Every time you look at a screw, a nail, a brick, a board ... all are manufactured to standard sizes. (NB: Some standards are "soft," e.g., the 2.54 cm nail standard.)

Computer manufacturers kind of have standards. Consider characters. I learned on a Bendix G-15. It had the extended character set option and could actually **print letters** instead of just numbers. This was a startling innovation (all the more startling due to its blazing speed – three characters/second!). The G-15 had 29-bit words and bizarre encoding for the characters. The coding scheme died a merciful death.

By the 60's, the ASCII code emerged. You know: the American Standard Code for Information Interchange. All American manufacturers were to adhere to it voluntarily. Every one of them. Except CDC, which was busy with 6-bit character codes, 10 per word. Except PLATO, CDC machines which had variable length characters (6 to 24 bits). Except UNIVAC; they used "field data," more 6-bit characters. Not too many special characters there, nosirree! Case distinctions? Who needs it! They also had the "quarter-word" format which stored four 9-bit characters. That was enough for upper/lower case and some exciting nonstandard graphics. Except DEC. Their DECsystem-10 had a scheme which encoded characters using a MOD-50 scheme. Innovative.

And: except IBM. They decided to use EBCDIC instead. Terrific. Instead of the ISO or any other standard, they had a new kind: the de facto standard. The phrase "de facto" means "everyone does it this way so it doesn't matter what you think." IBM is real big on de facto standards.

Time passed; the world turned around once every day. Soon people found that the fewer ways there were to do a given thing (e.g., character codes) the more productive they could be. The world of computers has seen many standards emerge in recent years. Early on, tape formats standardized, thus enhancing interchange of data among various systems. Local area networks fueled the need for standards as each manufacturer found they needed to meet some level of compatibility or die. The personal computer world has almost achieved the world of plug-and-play for some kinds of peripherals and computers. What an amazing universe we now live in.

So what's the complaint? I'll tell you the complaint: the very word "standard" is now bantied about as if it means "latest way we invented to do something." When's the last time YOU said, "Oh, I think I'll invent a new page-description language; we can make it a standard!" Or maybe: "Gosh, I don't think there's enough network file systems in the world; let's have a NEW STANDARD." AT&T tried that one. Oops.

Standards are hard. Either you have to let one guy (maybe two if they're friends) do it or you have to have a "committee." Committees can do it: it's been proven. Unfortunately, they take longer. The last FORTRAN standard took 10 years. The next one, currently dubbed FORTRAN 8x may not make it! It may turn out to be FORTRAN 9x. How disappointing.

. . . *continued*

At any rate, while companies like IBM can create de facto standards just because they sell some substantial fraction of every computer in the world, that doesn't mean just anyone can.

Let's all see how we can cooperate in the coming year and have just a few standards – a few good ones.

≡ ≡ ≡

Call for Papers – System Administration Workshop

USENIX director Rob Kolstad will chair a Large Installation System Administrator's Workshop to be held in Philadelphia, Pa., on Thursday and Friday, April 9th and 10th, 1987. The workshop will bring together system administrators trying to conquer UNIX's historical bias towards smaller systems. It is believed these administrators solve many of the same problems repeatedly and can share their unique solutions to some problems in order to avoid duplication of effort as UNIX grows to run in ever-larger installations. System managers of shops with over 100 users (on one or several processors) will find this workshop particularly valuable. It is intended that each administrator will briefly present unique attributes of his/her environment and contribute a short (five minute) discussion (and paper) detailing some solution from his/her shop.

Some topics to be considered include: large file systems (dumps, networked file systems), password file administration (including YP), large mail system administration, Usenet/News/Notes administration, mixed vendor (and version) environments, load control and batch systems, handy new utilities, and large LANs.

For participation information, please contact

Rob Kolstad
Convex Computer Corporation
701 Plano Road
Richardson, TX 75081
(214) 952-0351
{sun,allegra,ihnp4}!convex!kolstad

Call Rob ASAP if you are interested in co-chairing this workshop.

Future Meetings

4th NZUSUGI Meeting – Auckland

The New Zealand UNIX Systems User Group Inc. will hold its fourth annual conference May 13-16, 1987, at the Travelodge at the Auckland International Airport. The theme of the conference is "UNIX as you like it." There will be pre-conference seminars and both commercial and technical streams.

For further information, contact:

Ian M. Howard
c/o CCL Business Systems Ltd.
PO Box 3323
Auckland, New Zealand

USENIX 1987 Summer Conference and Exhibition – Phoenix

The USENIX 1987 Summer Conference and Exhibition will be held on June 8-12, 1987, at the Hyatt Regency Hotel in Phoenix, Arizona. There will be a conference, tutorials, and vendor exhibits. Please see the Call for Papers on page 3 of this issue of *;login:*.

USENIX 1988 Winter Conference and UniForum – Dallas

The USENIX 1988 Winter Conference will be held on February 10-12, 1988, at the Registry Hotel in Dallas, Texas. It will be concurrent with UniForum 1988, which will also be in Dallas. The Conference will feature tutorials and technical sessions.

USENIX 1988 Summer Conference and Exhibition – San Francisco

The USENIX 1988 Summer Conference and Exhibition will be held on June 21-24, 1988, at the Hilton Hotel in San Francisco, California. There will be a conference, tutorials, and vendor exhibits.

Long-term USENIX Conference Schedule

Jun 8-12 '87 Hyatt Regency, Phoenix, AZ
Feb 10-12 '88 Registry Hotel, Dallas, TX
Jun 21-24 '88 Hilton Hotel, San Francisco, CA
Feb 1- 3 '89 San Diego, CA
Jun 13-16 '89 Hyatt Regency, Baltimore, MD
Jun 11-15 '90 Marriott Hotel, Anaheim, CA

≡ ≡ ≡

Atlanta Videotapes Available

The three presentations which opened the Atlanta USENIX Conference were videotaped. Orders for the tapes are now being accepted.

There are two tapes:

Tape one contains Jon Bentley's "Pictures of Programs," a talk on the merits and insights of graphic presentation of algorithmic problems, including the greedy travelling salesman and a wire-wrapping problem. In addition, this tape contains the Awards Ceremonies and Presentations.

Tape two contains two music presentations: Mike Hawley's "MIDI Music Software for UNIX" and Peter Langston's "(201) 644-2332 or Eedie and Eddie on the wire: An experiment in Music Generation."

Each tape is available in either VHS or Beta format. Inside the US and Canada, the tapes are $20 each, including postage and handling charges. Elsewhere in the world, the tapes are $30 each, including air mail charges.

Place orders directly with USENIX Association, including a check or money order in US dollars for the full amount due. Please remember to specify which tape(s) you want and which format you desire. Allow 4 to 6 weeks for delivery.

– PHS

4.3BSD UNIX Manuals

The USENIX Association is sponsoring production of the 4.3BSD UNIX Manuals for its Institutional and Supporting members†. This article provides information on the contents, cost, and ordering of the manuals.

The 4.3BSD manual sets are significantly different from the 4.2BSD edition. Changes include many additional documents, better quality of reproductions, as well as a new and extensive index. All manuals are printed in a photo-reduced 6″×9″ format with individually colored and labeled plastic "GBC" bindings. All documents and manual pages have been freshly typeset and all manuals have "bleed tabs" and page headers and numbers to aid in the location of individual documents and manual sections.

A new Master Index has been created. It contains cross-references to all documents and manual pages contained within the other six volumes. The index was prepared with the aid of an "intelligent" automated indexing program from Thinking Machines Corp. along with considerable human intervention from Mark Seiden. Key words, phrases and concepts are referenced by abbreviated document name and page number.

While two of the manual sets contain three separate volumes, you may only order complete sets.

The costs shown below do not include applicable taxes or handling and shipping from the publisher in New Jersey, which will depend on the quantity ordered and the distance shipped. Those charges will be billed by the publisher (Howard Press).

Manuals are available now. To order, return a completed "4.3BSD Manual Reproduction Authorization and Order Form" to the USENIX office along with a check or purchase order for the cost of the manuals. You **must** be a USENIX Association Institutional or Supporting member. Checks and purchase orders should be made out to Howard Press. Orders will be forwarded to the publisher after license verification has been completed, and the manuals will be shipped to you directly from the publisher.

Manual	Cost*
User's Manual Set (3 volumes) User's Reference Manual User's Supplementary Documents Master Index	$25.00/set
Programmer's Manual Set (3 volumes) Programmer's Reference Manual Programmer's Supplementary Documents, Volume 1 Programmer's Supplementary Documents, Volume 2	$25.00/set
System Manager's Manual (1 volume)	$10.00

* Not including postage and handling or applicable taxes.

4.2BSD Programmer's Manual Sets are still available at $17

† Tom Ferrin of the University of California at San Francisco, a former member of the Board of Directors of the USENIX Association, has overseen the production of the 4.2 and 4.3BSD manuals.

4.3BSD Manual Reproduction Authorization and Order Form

This page may be duplicated for use as an order form

USENIX Member No.: _____

Purchase Order No.: _____

Date: _____

As the representative of a USENIX Association Institutional or Supporting Member in good standing, and as a *bona fide* license holder of both a 4.3BSD software license from the Regents of the University of California and a UNIX®/32V or System III or System V license or sublicense from AT&T and pursuant to the copyright notice as found on the rear of the cover page of the UNIX®/32V Programmer's Manual stating that

> "Holders of a UNIX®/32V software license are permitted to copy this document, or any portion of it, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included,"

I hereby appoint the USENIX Association as my agent, to act on my behalf to duplicate and provide me with such copies of the Berkeley 4.3BSD Manuals as I may request.

Signed (authorized Institutional representative): _____

Institution: _____

Ship to:

Name: _____

_____

_____

_____

_____

Phone: _____

Billing address, if different:

Name: _____

_____

_____

_____

_____

Phone: _____

The prices below **do not** include shipping and handling charges or state or local taxes. All payments must be in US dollars drawn on a US bank.

4.3BSD User's Manual Set (3 vols.)	_____	at $25.00 each	= $_____
4.3BSD Programmer's Manual Set (3 vols.)	_____	at $25.00 each	= $_____
4.3BSD System Manager's Manual (1 vol.)	_____	at $10.00 each	= $_____
Total	_____		$_____

[ ] Purchase order enclosed; invoice required.
(Purchase orders **must** be enclosed with this order form.)

[ ] Check enclosed for the manuals: $_____
(Howard Press will send an invoice for the shipping and handling charges and applicable taxes.)

Make your check or purchase order out to Howard Press and mail it with this order form to:

Howard Press
c/o USENIX Association
P.O. Box 7
El Cerrito, CA 94530

for office use: l.v.: _____ check no.: _____ amt. rec'd: _____

Proceedings of 3rd Graphics Workshop Available

The Proceedings of the Third Computer Graphics Workshop are now available. The Conference, held on November 20-21, 1985, in Monterey was a great success and the 160-page Proceedings volume contains a number of stunning papers. The full contents are:

The Utah Raster Toolkit
John W. Peterson, Rod G. Bogart, and Spencer W. Thomas, University of Utah

A High-End High-Performance Graphics System for Computational Fluid Dynamics
Julian E. Gomez, Research Institute for Advanced Computer Science, *Frank Preston,* National Aeronautics and Space Administration, *Steve Fine, Tony Hasegawa, Bock Lee, and Blaine Walker,* General Electric Western Systems, Space Systems Division

Recollections
Ed Tannenbaum, Eddeo Inc.

Pictorial Conversation: Design Considerations for Interactive Graphical Media
Rob Myers, Silicon Graphics Computer Systems

Plasm: A Fish Sample
Rob Myers, Peter Broadwell, and Robin Schaufler, Silicon Graphics Computer Systems

A Multi-Representation, Bitmap Interface to the UNIX File System Constructed from Cooperating Processes
C. D. Blewett, J. T. Edmark, J. I. Helfman, and M. Wish, AT&T Bell Laboratories

Scattered Thoughts on B-splines
Spencer W. Thomas, University of Utah

Procedural Spline Interpolation in UNICUBIX
Carlo H. Séquin, University of California, Berkeley

Porting UNIX to the Bösendorfer
Michael Hawley, Massachusetts Institute of Technology

A Low Cost, Video Based, Animated Movie System for the Display of Time Dependent Modeling Results
William E. Johnston, Dennis E. Hall, Fritz Renema, and David Robertson, Lawrence Berkeley Laboratory

Object Oriented Programming in NeWS
Owen M. Densmore, Sun Microsystems

Computer Assisted Color Conversion
David M. Geshwind, Digital Video Systems

Copies of the Proceedings are available from the USENIX Association at $10.00 each (overseas surface postage, $5; overseas air postage, $15), prepaid.

There will be a Fourth Graphics Workshop in 1987; see page 12 of this issue of **;login:** *for the Call for Papers.*

Publications Available

The following publications are available from the Association Office or the source indicated. Prices and overseas postage charges are per copy. California residents please add applicable sales tax. Payments **must** be enclosed with the order and **must** be in US dollars payable on a US bank.

USENIX Conference and Workshop Proceedings

Meeting	Location	Date	Price	Overseas Mail Air	Overseas Mail Surface	Source
USENIX	**Wash. DC**	**Winter '87**	$20	$25	$5	USENIX
Graphics Workshop III	**Monterey**	**December '86**	$10	$15	$5	USENIX
USENIX	Atlanta	Summer '86	$25	$25	$5	USENIX
Graphics Workshop II	Monterey	December '85	$ 3	$ 7	$5	USENIX
USENIX	Denver	Winter '86	$20	$25	$5	USENIX
USENIX	Portland	Summer '85	$25	$25	$5	USENIX
USENIX	Dallas	Winter '85	$20	$25	$5	USENIX
Graphics Workshop I	Monterey	December '84	$ 3	$ 7	$5	USENIX
USENIX	Salt Lake	Summer '84	$25	$25	$5	USENIX
UniForum	Wash. DC	Winter '84	$30	$20		/usr/group

EUUG Publications

The following EUUG publications may be ordered from the USENIX Association office.

The EUUG Newsletter, which is published four times a year, is available for $4 per copy or $16 for a full-year subscription. The earliest issue available is Volume 3, Number 4 (Winter 1983).

The July 1983 edition of the EUUG Micros Catalog is available for $8 per copy.

≡ ≡ ≡

Human – Computer Interaction Conference

A major North American conference focusing on the improvement of interaction between humans and computers will be held in Toronto, April 5-9, 1987.

The Conference is a combination of CHI '87 (Human Factors in Computing Systems) and GI '87 (Graphics Interface). The annual CHI conference (sponsored by the ACM (Association for Computing Machinery) Special Interest Group on Computers and Human Interaction, SIGCHI) is the leading forum for the presentation of original designs and research in all aspects of human-computer interaction.

Twenty five tutorials will be offered by internationally known specialists.

For information, contact:

Wendy Walker
Conference Coordinator
CHI + GI '87 Conference Office
Computer Systems Research Institute
University of Toronto
10 Kings College Road, Room 2002
Toronto, Ont. CANADA M5S 1A4
(416) 978-5184
Walker-CHI%toronto@CSNET-RELAY

Local User Groups

The USENIX Association will support local user groups in the following ways:

• Assisting the formation of a local user group by doing an initial mailing for the group. This mailing may consist of a list supplied by the group, or may be derived from the USENIX membership list for the geographical area involved. At least one member of the organizing group must be a current member of the USENIX Association. Membership in the group must be open to the public.

• Publishing information on local user groups in *;login:* giving the name, address, phone number, net address, time and location of meetings, etc. Announcements of special events are welcome; send them to the editor at the USENIX office.

Please contact the USENIX office if you need assistance in either of the above matters. Our current list of local groups follows.

In the **Atlanta** area there is a group for people with interest in UNIX or UNIX-like systems, which meets on the first Monday of each month in White Hall, Emory University.

Atlanta UNIX Users Group
P.O. Box 12241
Atlanta, GA 30355-2241

Marc Merlin (404) 442-4772
Mark Landry (404) 365-8108

In the **Boulder**, Colorado area a group meets about every two months at different sites for informal discussions.

Front Range Users Group
USENIX Association Exhibit Office
Oak Bay Building
4750 Table Mesa Drive
Boulder, CO 80303

John L. Donnelly (303) 499-2600
usenix!johnd

A UNIX users group has formed in the **Coral Springs**, Florida, area. For information, contact:

S. Shaw McQuinn (305) 344-8686
8557 W. Sample Road
Coral Springs, FL 33065

Dallas / Fort Worth UNIX User's Group
Seny Systems, Inc.
5327 N. Central, #320
Dallas, TX 75205

Jim Hummel (214) 522-2324

In **East Lansing**, Michigan, the Michigan State University UNIX Users Group meets approximately monthly. It is open to all interested persons, not only University affiliates. For meeting times and more information, contact:

John H. Lawitzke (517) 355-3769
Division of Engineering Research
364 Engineering Building
Michigan State University
East Lansing, MI 48824

ihnp4!msudoc!eecae!lawitzke

The **Los Angeles** UNIX Group (LAUG) meets on the third Thursday of each month in Redondo Beach, California. For additional information, please contact:

Drew Bullard (213) 535-1980
{ucbvax,ihnp4}!trwrb!bullard

Marc Ries (213) 535-1980
{decvax,sdcrdcf}!trwrb!ries

In **Minnesota** a group meets on the first Wednesday of each month. For information, contact:

UNIX Users of Minnesota
Shane McCarron (612) 786-1496
ihnp4!meccts!ahby or ahby@mecc.com

In the northern **New England** area is a group that meets monthly at different sites. Contact one of the following for information:

Emily Bryant (603) 646-2999
Kiewit Computation Center
Dartmouth College
Hanover, NH 03755

decvax!dartvax!emilyb

David Marston (603) 883-3556
Daniel Webster College
University Drive
Nashua, NH 03063

In the **New York City** area there is a non-profit organization for users and vendors of products and services for UNIX systems.

Unigroup of New York
G.P.O. Box 1931
New York, NY 10116

Ed Taylor (212) 513-7777
{attunix,philabs}!pencom!taylor

The **New Zealand** group provides an annual Workshop and Exhibition and a regular newsletter to its members.

New Zealand UNIX Systems User Group
P.O. Box 13056
University of Waikato
Hamilton, New Zealand

An informal group has started in the **St. Louis** area:

St. Louis UNIX Users Group
Plus Five Computer Services
765 Westwood, 10A
Clayton, MO 63105

Eric Kiebler (314) 725-9492
ihnp4!plus5!sluug

In the **San Antonio** area the San Antonio UNIX Users (SATUU) meet twice each month with the second Wednesday being a dinner meeting and the third Wednesday being a "roving" meeting at a user site.

San Antonio UNIX Users
7950 Floyd Curl Dr. #102
San Antonio, TX 78229-3955

William T. Blessum, M.D. (512) 692-0977
ihnp4!petro!bles!wtb

In the **Seattle** area there is a group with over 150 members, a monthly newsletter, and a software exchange system. Meetings are held monthly.

Bill Campbell (206) 232-4164
Seattle UNIX Group Membership Information
6641 East Mercer Way
Mercer Island, WA 98040

uw-beaver!tikal!camco!bill

A UNIX/C language users group has been formed in **Tulsa**. For current information on meetings, etc. contact:

Pete Rourke
$USR
7340 East 25th Place
Tulsa, OK 74129

USENIX Association
P.O. Box 7
El Cerrito, CA 94530

First Class Mail

Call for Papers – Summer USENIX Conference

Call for Papers – System Administrator's Workshop

Call for Papers – 4th Computer Graphics Workshop

How to Write a Setuid Program

An Overview of the Sprite Project

Change of Address Form

Please fill out and send the following form through the U.S. mail to the Association Office at the address above.

Name: _____ Member #: _____

OLD: _____

_____

_____

_____

NEW: _____

_____

_____

_____

Phone: _____

uucp: {decvax,ucbvax}! _____